

# *Modelar con Java*

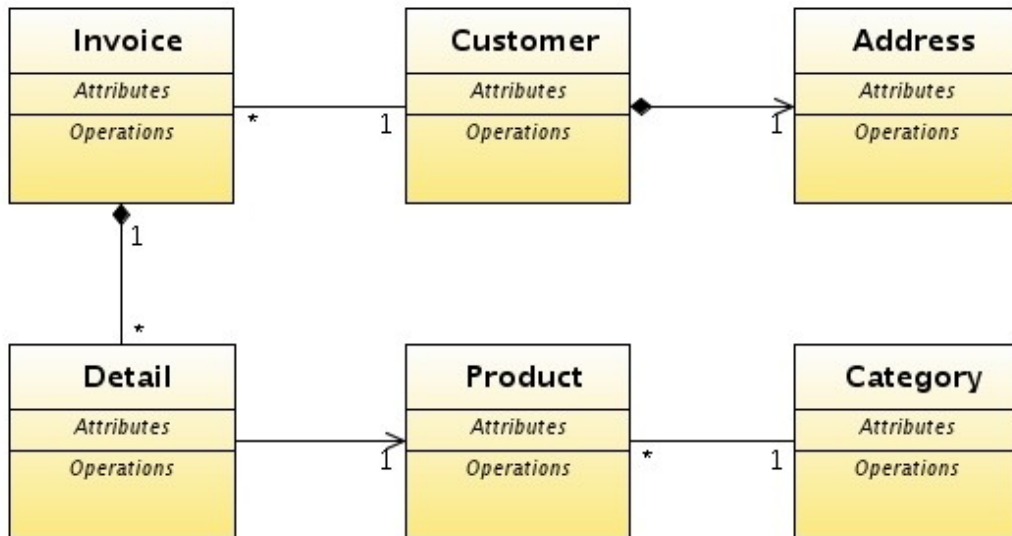
## lección2

Ahora que tienes tu entorno configurado y sabes como desarrollar con él, es hora de dar forma a tu proyecto. En esta lección, crearás todas las entidades de tu proyecto y tendrás tu aplicación funcionando en un santiamén.

Asumo que sabes crear una nueva entidad con Eclipse, como actualizar el esquema de la base de datos cada vez que cambias tus entidades y como ejecutar la aplicación, porque ya has leído la lección 1.

## 2.1 Modelo básico del dominio

Primero crearemos las entidades para tu aplicación Invoicing. El modelo del dominio es más bien básico, pero suficiente para aprender bastantes cosas interesantes. Lo puedes ver en la figura 2.1.



**Figura 2.1** Diagrama UML inicial para la aplicación Invoicing

Empezaremos con seis clases, y más adelante añadiremos algunas más. Recuerda que ya tienes una versión inicial de Customer y Product.

### 2.1.1 Referencia (ManyToOne) como lista de descripciones (combo)

Empecemos con el caso más simple. Vamos a crear una entidad Category y asociarla a Product, visualizándola con un combo.

El código para la entidad Category está en el listado 2.1.

## 21 Lección 2: Modelar con Java

### Listado 2.1 Entidad Category con generación de oid UUID

```
package org.openxava.invoicing.model;

import javax.persistence.*;

import org.hibernate.annotations.GenericGenerator;
import org.openxava.annotations.*;

@Entity
public class Category {

    @Id
    @Hidden // La propiedad no se muestra al usuario. Es un identificador interno
    @GeneratedValue(generator="system-uuid") // Identificador Universal Único (1)
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid;

    @Column(length=50)
    private String description;

    // Getters y setters
    ...

}
```

Sin duda, la entidad más simple posible. Solo tiene un identificador y una propiedad `description`. En este caso usamos el algoritmo Identificador Universal Único (1) para generar el identificador. La ventaja de este generador de identificadores es que puedes migrar tu aplicación a otras bases de datos (DB2, MySQL, Oracle, Informix, etc) sin tocar tu código. Los otros generadores de identificadores de JPA usan la base de datos para generar el identificador, por lo que no son tan portables como UUID.

Ahora puedes ejecutar el módulo `Category` y añadir algunas categorías. Recuerda actualizar el esquema de la base de datos primero.

Ahora, asociaremos `Product` con `Category`. Míralo en el listado 2.2.

### Listado 2.2 Product con una referencia a Category

```
@Entity
public class Product {

    @Id @Column(length=9)
    private int number;

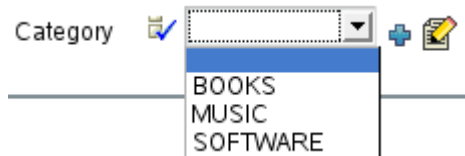
    @Column(length=50) @Required
    private String description;

    @ManyToOne( // La referencia se almacena como una relación en la base de datos
        fetch=FetchType.LAZY, // La referencia se carga bajo demanda
        optional=true) // La referencia puede estar sin valor
    @DescriptionsList // Así la referencia se visualiza usando un combo
    private Category category; // Una referencia Java convencional

}
```

```
// Getters y setters
...
}
```

Es una simple relación muchos-a-uno de JPA, como se puede ver en el apéndice B. En este caso, gracias a la anotación `@DescriptionsList` se visualiza usando un combo (figura 2.2).



**Figura 2.2 Referencia a Category visualizada como combo**

Ahora es el momento de completar la entidad `Product`.

### 2.1.2 Estereotipos

La entidad `Product` necesita tener al menos precio, además estaría bien que tuviese fotos y un campo para observaciones. Vamos a usar estereotipos para conseguirlo. Un estereotipo especifica un uso específico de un tipo. Por ejemplo, puedes usar `String` para almacenar nombres, comentarios o identificadores, y puedes usar `BigDecimal` para almacenar porcentajes, dinero o cantidades. Es decir, hacemos diferentes usos del mismo tipo. Los estereotipo son justo para marcar este uso específico.

La mejor forma de entender que es un estereotipo es verlo en acción. Añadamos las propiedades `price`, `photo`, `morePhotos` y `remarks` a tu entidad `Product` (listado 2.3).

#### Listado 2.3 Nuevas propiedades para `Product` que usan `@Stereotype`

```
@Stereotype("MONEY") // La propiedad price se usa para almacenar dinero
private BigDecimal price; // BigDecimal se suele usar para dinero

@Stereotype("PHOTO") // El usuario puede ver y cambiar una foto
private byte [] photo;

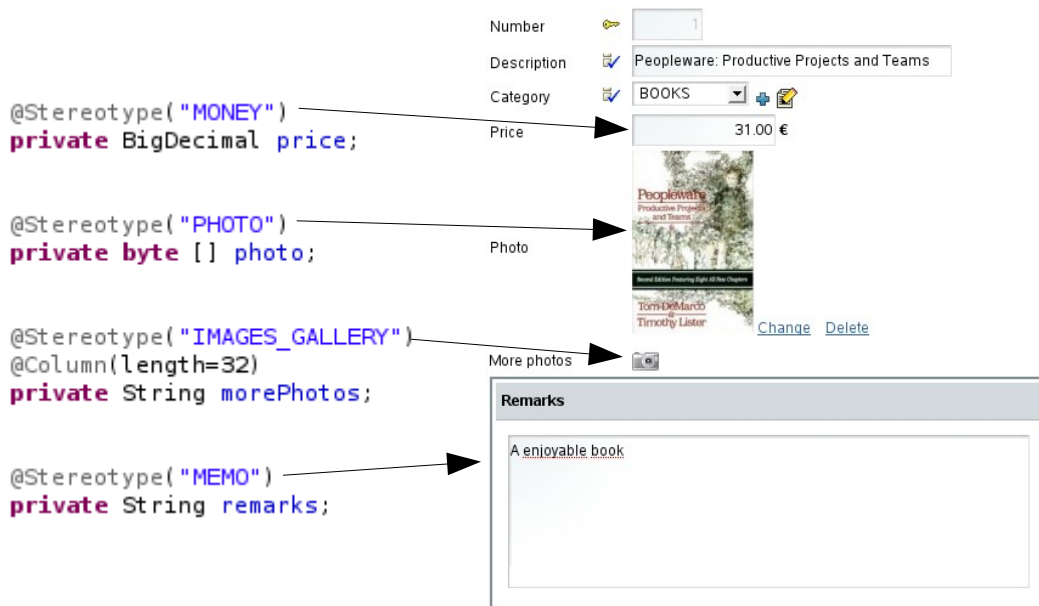
@Stereotype("IMAGES_GALLERY") // Una galería de fotos completa está disponible
@Column(length=32) // La cadena de 32 de longitud es para almacenar la clave de la galería
private String morePhotos;

@Stereotype("MEMO") // Esto es para un texto grande, se usará un área de texto o equivalente
private String remarks;

// Getters y setters
```

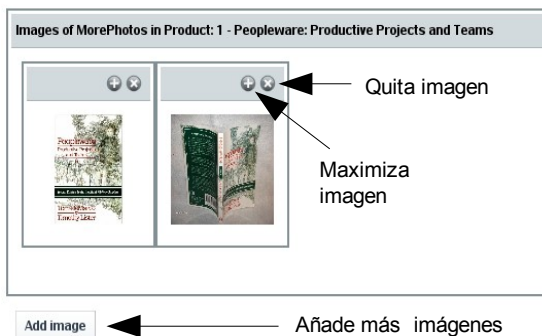
Has visto como usar estereotipos, solo has de poner el nombre del estereotipo y OpenXava hará un tratamiento especial. Si ejecutas el módulo para `Product` ahora verás lo que hay en la figura 2.3.

## 23 Lección 2: Modelar con Java



**Figura 2.3 Efecto visual de los esterotipos en la interfaz de usuario**

Como puedes ver en la figura 2.3, cada estereotipo produce un efecto en la interfaz de usuario. Los estereotipos tienen efecto en los tamaños, validaciones, editores, etc. Y te permiten reutilizar funcionalidad predefinida con facilidad. Por ejemplo, sólo marcando una simple propiedad String con `@Stereotype("IMAGES_GALLERY")` tendrás disponible toda una galería de imágenes. Pulsando en la cámara de la propiedad `morePhotos` verás la galería de la figura 2.4.



**Figura 2.4 Galería de imágenes producida por el estereotipo IMAGE\_GALLERY**

A parte de estos, OpenXava tiene muchos estereotipos predefinidos que te pueden ser útiles, tales como LABEL, BOLD\_LABEL, DATETIME, ZERO\_FILLED, HTML\_TEXT, IMAGE\_LABEL, EMAIL, TELEPHONE, WEBURL, IP, ISBN, CREDIT\_CARD, EMAIL\_LIST.

Ya tienes Product listo. Refinemos ahora Customer.

### 2.1.3 Embeddable

Vamos a añadir una dirección (address) a nuestro, hasta ahora algo desnudo,

Customer. La dirección del Customer no está compartida por otros objetos Customer, y cuando un Customer se borra, su dirección (address) es borrada también, por lo tanto modelaremos el concepto de dirección como una clase incrustable. Este se puede ver en el apéndice B sobre JPA.

Añade la clase Address a tu proyecto. Su código está en listado 2.4.

#### Listado 2.4 Address se ha modelado como una clase incrustable (embeddable)

```
@Embeddable // Usamos @Embeddable en vez de @Entity
public class Address {

    @Column(length=30) // Los miembros se anotan igual que en las entidades
    private String street;

    @Column(length=5)
    private int zipCode;

    @Column(length=20)
    private String city;

    @Column(length=30)
    private String state;

    // Getters y setters
    ...
}
```

Como ves, es una clase normal y corriente anotada como @Embeddable. Sus propiedades se anotan de la misma manera que en las entidades, aunque las clases incrustables no soportan toda la funcionalidad de las entidades.

Ahora, puedes usar Address en cualquier entidad. Simplemente añade una referencia a ella en tu entidad Customer, dejándola como en el listado 2.5.

#### Listado 2.5 La entidad Customer con una referencia a la incrustable Address

```
@Entity
public class Customer {

    @Id
    @Column(length=6)
    private int number;

    @Column(length=50)
    @Required
    private String name;

    @Embedded // Así para referenciar a una clase incrustable
    private Address address; // Una referencia Java convencional

    public Address getAddress() {
        if (address == null) address = new Address(); // Así nunca es nulo
        return address;
    }
}
```

## 25 Lección 2: Modelar con Java

```
public void setAddress(Address address) {  
    this.address = address;  
}  
  
// Otros getters y setters  
...  
}
```

Los datos de Address se almacenan en la misma tabla que los de Customer. Y desde una perspectiva de la interfaz de usuario hay un marco alrededor de Address, aunque si no te gusta el marco solo has de anotar la referencia con @NoFrame, como muestra el listado 2.6.

### Listado 2.6 Address con @NoFrame

```
@Embedded @NoFrame // Con @NoFrame no se muestra marco para address  
private Address address;
```

La figura 2.5 muestra la interfaz de usuario para una referencia incrustada con y sin @NoFrame.

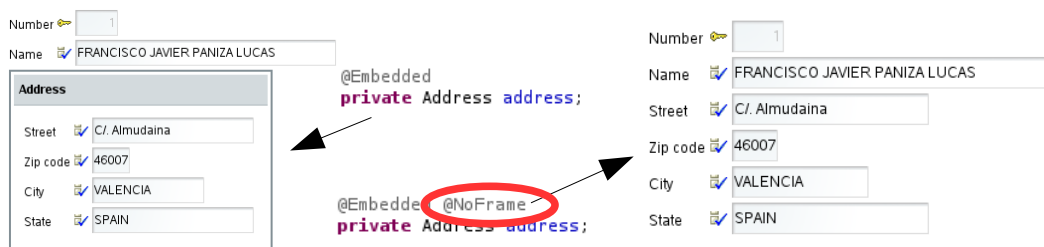


Figura 2.5 Interfaz de usuario con referencias incrustables con y sin @NoFrame

Ahora que tenemos las entidades básicas en marcha, es el momento de enfrentarnos a la entidad principal de la aplicación: Invoice. Empecemos poco a poco.

### 2.1.4 Clave compuesta

No vamos a usar una clave compuesta para Invoice. Es mejor evitar el uso de claves compuestas. Siempre tienes la opción de usar un identificador oculto autogenerado. Aunque, algunas veces tienes la necesidad de conectarte a bases de datos legadas o puede que el diseño del esquema lo haya hecho alguien que le gustan las claves compuestas, y no tengas otra opción que usar claves compuestas aunque no sea lo ideal. Por lo tanto, vamos a aprender como usar una clave compuesta, aunque al final cambiaremos a una clave simple autogenerada.

Empecemos con una versión sencilla de la entidad Invoice. Mírala en el listado 2.7.

**Listado 2.7 Primera versión de Invoice, con clave compuesta**

```

@Entity
@IdClass(InvoiceKey.class) // La clase id contiene todas las propiedades clave (1)
public class Invoice {

    @Id // Aunque tenemos la clase id aún es necesario marcarlo como @Id (2)
    @Column(length=4)
    private int year;

    @Id // Aunque tenemos la clase id aún es necesario marcarlo como @Id (2)
    @Column(length=6)
    private int number;

    @Required
    private Date date;

    @Stereotype("MEMO")
    private String remarks;

    // Getters y setters

}

```

Si quieres usar year y number como clave compuesta para Invoice, una forma de hacerlo, es marcándolos con @Id (2), y además tener una clase id (1). La clase id tiene que tener year y number como propiedades. Puedes verla en el listado 2.8.

**Listado 2.8 InvoiceKey: La clase id para Invoice**

```

public class InvoiceKey
    implements java.io.Serializable { // La clase key tiene que ser serializable

    private int year; // Contiene las propiedades marcadas ...
    private int number; // ... como @Id en la entidad

    @Override
    public boolean equals(Object obj) { // Ha de definir el método equals
        if (obj == null) return false;
        return obj.toString().equals(this.toString());
    }

    @Override
    public int hashCode() { // Ha de definir el método hashCode
        return toString().hashCode();
    }

    @Override
    public String toString() {
        return "InvoiceKey::" + year + ":" + number;
    }

    // Getters y setters para year y number

}

```

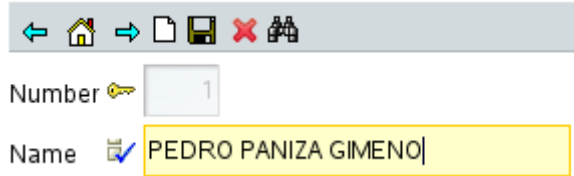
En el listado 2.8 se ven algunos de los requerimientos para una clase id, como



## 27 Lección 2: Modelar con Java

el ser serializable e implementar hashCode() y equals().

Has visto como usar una clave compuesta. Pero dado que tenemos control sobre nuestro esquema, al final vamos a usar un identificador UUID para Invoice. Reescribe la entidad Invoice y déjala como en el listado 2.9.



**Figura 2.2** Interfaz de usuario para la entidad Customer con 2 propiedades

### Listado 2.9 Invoice usando una clave simple con UUID

```
package org.openxava.invoicing.model;

import java.util.*;
import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
import org.openxava.annotations.*;

@Entity // @IdClass eliminada
public class Invoice {

    @Id @GeneratedValue(generator="system-uuid") @Hidden
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid; // Añadida como clave oculta autogenerada
                       // Acuerdate de añadir getOid() y setOid()

    @Column(length=4) // @Id quitado
    @DefaultValueCalculator(CurrentYearCalculator.class)
    private int year;

    @Column(length=6) // @Id quitado
    @DefaultValueCalculator(value=NextNumberForYearCalculator.class,
        properties=@PropertyValue(name="year")
    )
    private int number;

    ...
}
```

También borra la clase InvoiceKey. Usar una clave oculta autogenerada para Invoice tiene varios beneficios prácticos sobre una clave compuesta: No has de escribir la aburrida InvoiceKey, puedes modificar el número de factura sin perder ninguna asociación con otros objetos y puedes almacenar en la misma tabla pedidos (*orders*) y facturas (*invoices*) con el par año/numero repetido.

El código que tienes es suficiente para hacer funcionar el módulo Invoice. Hazlo y añade algunas facturas si quieres. Aunque todavía queda mucho trabajo por hacer en Invoice, como asignar los valores por defecto para year, number y date.

### 2.1.5 Calcular valores por defecto

Si has probado el módulo Invoice, habrás visto que necesitas teclear el año, el número y la fecha. Estaría bien tener valor por defecto. Es fácil de hacer usando la anotación `@DefaultValueCalculator`. En el listado 2.10 ves como podemos añadir los valores por defecto para year y date:

**Listado 2.10** Calculadores valor defecto incluidos en OpenXava para año y fecha

```
@Column(length=4)
@DefaultValueCalculator(CurrentYearCalculator.class) // Año actual
private int year;

@Required
@DefaultValueCalculator(CurrentDateCalculator.class) // Fecha actual
private Date date;
```

A partir de ahora cuando el usuario pulse en el botón 'nuevo' el campo para año será el año actual, y el campo para la fecha la fecha actual. Estos dos calculadores (`CurrentYearCalculator` y `CurrentDateCalculator`) están incluidos en OpenXava. Explora el paquete `org.openxava.calculators` para ver otros calculadores predefinidos que pueden serte útiles.

Pero a veces necesitas tu propia lógica para calcular el valor por defecto. Por ejemplo, para number queremos sumar uno al último número de factura dentro de este mismo año. Crear tu propio calculador con tu lógica es fácil. Primero, crea un paquete `org.openxava.invoicing.calculators` para los calculadores. Y crea en él una clase `NextNumberForYearCalculator`, con el código del listado 2.11.

**Listado 2.11** Calculador propio para el valor por defecto del número de factura

```
package org.openxava.invoicing.calculators;

import javax.persistence.*;
import org.openxava.calculators.*;
import org.openxava.jpa.*;

public class NextNumberForYearCalculator
    implements ICalculator { // Un calculador tiene que implementar ICalculator

    private int year; // Este valor se inyectará (usando su setter) antes de calcular

    public Object calculate() throws Exception { // Hace el cálculo
        Query query = XPersistence.getManager() // Una consulta JPA
            .createQuery("select max(i.number) from Invoice i" +
                " where i.year = :year"); // La consulta devuelve el número de factura
                                           // máximo del año indicado
        query.setParameter("year", year); // Ponemos el año inyectado como parámetro
                                           // de la consulta
        Integer lastNumber = (Integer) query.getSingleResult();
        return lastNumber == null ? 1 : lastNumber + 1; // Devuelve el último número
                                                         // de factura del año + 1
                                                         // o 1 si no hay último número
    }
}
```

```

    }

    public int getYear() {
        return year;
    }

    public void setYear(int year) {
        this.year = year;
    }
}

```

Tu calculador tiene que implementar `ICalculator` (y por lo tanto tener un método `calculate()`). Declaramos una propiedad `year` para poner en ella el año del cálculo. Para implementar la lógica usamos una consulta JPA. Repásate el apéndice B sobre JPA. Ahora solo queda anotar la propiedad `number` en la entidad `Invoice` (listado 2.12).

#### Listado 2.12 Propiedad `number` de `Invoice` anotada con un calculador propio

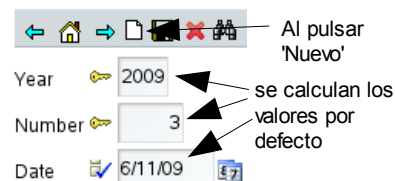
```

@Id @Column(length=6)
@DefaultValueCalculator(value=NextNumberForYearCalculator.class,
    properties=@PropertyValue(name="year") // Para inyectar el valor de year de Invoice
                                           // en el calculador antes de llamar a calculate()
)
private int number;

```

En este caso ves algo nuevo, `@PropertyValue`. Usándolo, estás diciendo que el valor de la propiedad `year` en la `Invoice` actual se moverá a la propiedad `year` del calculador antes de hacer el cálculo. Ahora cuando el usuario pulse en 'nuevo' el siguiente número de factura disponible para este año estará en el campo. La forma de calcular el número de factura no es el mejor para muchos usuarios concurrentes añadiendo facturas. No te preocupes, lo mejoraremos más adelante.

Puedes ver el efecto visual del calculador para valor por defecto en la figura 2.6. Los valores por defecto son solo los valores iniciales, el usuario los puede cambiar si así lo desea.



**Figura 2.6 Efecto de los calculadores de valores por defecto**

### 2.1.6 Referencias convencionales (*ManyToOne*)

Ahora que tenemos todas las propiedades atómicas listas para usar, es tiempo de añadir relaciones con otras entidades. Empezaremos añadiendo una referencia desde `Invoice` a `Customer`, porque una factura sin cliente no parece demasiado útil. Añade el código del listado 2.13 a la entidad `Invoice`.

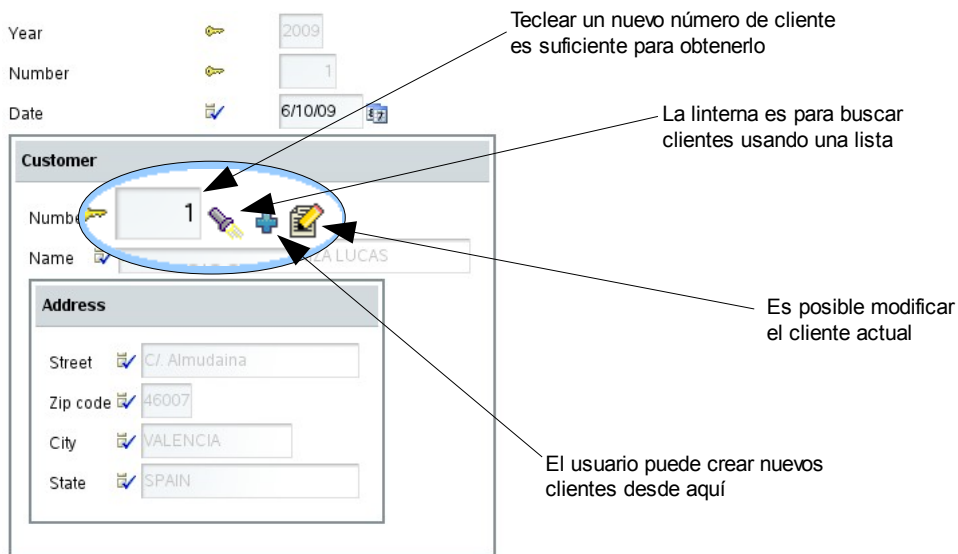
**Listado 2.13 Referencia a Customer desde Invoice**

```
@ManyToOne(fetch=FetchType.LAZY, optional=false) // customer es obligatorio
private Customer customer;

// Getters y setters para customer
```

No hace falta más. Sin embargo, antes de probar el nuevo módulo Invoice has de borrar las facturas actuales (ejecutando 'DELETE FROM Invoice'), y entonces actualizar el esquema. Borrar las facturas que hemos añadidos es necesario porque optional=false de la anotación @ManyToOne no permite facturas sin cliente<sup>2</sup>.

El módulo Invoice es como lo que se muestra en la figura 2.7.



**Figura 2.7** Interfaz de usuario de la referencia customer en Invoice

No hay más trabajo que hacer aquí. Añadamos una colección de líneas de detalle a Invoice.

### 2.1.7 Colección de entidades dependientes (ManyToOne con cascade)

Usualmente una factura necesita tener varias líneas con productos, cantidades, etc. Estos detalles son parte de la factura, no son compartidos por otras facturas y cuando una factura se borra sus líneas de detalle son borradas con ella. Por tanto, la forma más natural de modelar los detalles de una factura es usando objetos incrustados. Por desgracia, JPA (al menos en la 1.0) no soporta colecciones de

<sup>2</sup> Si quieres preservar los datos actuales pon optional=true, actualiza el esquema, actualiza los datos de cliente en las facturas existentes y pon optional=false de nuevo

### 31 Lección 2: Modelar con Java

objetos incrustables. Aunque la verdad es que esto no es un gran problema, porque puedes usar colecciones de entidades con el tipo de cascada REMOVE o ALL. Veámoslo en el caso de la definición de details en la entidad Invoice en el listado 2.14.

#### Listado 2.14 Colección details en Invoice

```
@OneToMany( // Para declararla como una colección persistente
    mappedBy="parent", // El miembro de Detail que almacena la relación
    cascade=CascadeType.ALL) // Indica que es una colección de entidades dependientes
private Collection<Detail> details = new ArrayList<Detail>();

// Getter y setter para details
```

Usando `cascade=CascadeType.ALL` cuando la factura se borra sus líneas se borran también. O si añades una nueva línea de detalle a una factura ya persistente, la línea es grabada automáticamente, y cuando marcas una factura como persistente sus líneas se marcan como persistentes también, y así por el estilo. Podemos decir que las entidades de la colección `details` son dependientes de `Invoice`. Esto es una forma bastante buena de simular usando entidades la semántica de los objetos incrustados. También puedes definir una colección dependiente con `cascade=CascadeType.REMOVE`.

Para que esta colección funcione necesitas escribir la clase `Detail` (listado 2.15).

#### Listado 2.15 Primera versión de la entidad Detail para Invoice

```
package org.openxava.invoicing.model;

import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
import org.openxava.annotations.*;

@Entity
public class Detail {

    @ManyToOne // Sin lazy fetching porque falla al quitar un detalle desde el padre
    private Invoice parent; // Así la relación entre Detail e Invoice es bidireccional

    @Id @GeneratedValue(generator="system-uuid") @Hidden
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid; // Identificador UUID, ver entidad Category (Listado 2.1)

    private int quantity;

    @ManyToOne(fetch=FetchType.LAZY, optional=true)
    private Product product;

    // Getters y setters
    ...
}
```

De momento solo tenemos quantity y product, pero es suficiente para tener la Invoice funcionando con details. Puedes ver en la figura 2.8 como el usuario puede añadir, editar y borrar elementos de la colección, filtrar y ordenar los datos y exportar a PDF y Excel.

Pero, la figura 2.8 enfatiza el hecho de que las propiedades a mostrar por defecto son las propiedades planas, es decir, las propiedades de las referencias no se incluyen por defecto. Este hecho produce una interfaz de usuario fea para nuestra colección de líneas de factura en nuestro caso, porque solo se muestra la propiedad quantity.

Por defecto muestra solo las propiedades planas de Detail  
En este caso solo la propiedad quantity

```
@OneToMany(mappedBy="parent", cascade=CascadeType.ALL)
private Collection<Detail> details;
```

Con @ListProperties el desarrollador puede especificar la propiedades a mostrar, incluyendo propiedades de referencia de cualquier nivel

```
@OneToMany(mappedBy="parent", cascade=CascadeType.ALL)
@ListProperties("product.number, product.description, quantity")
private Collection<Detail> details;
```

El usuario siempre tiene la opción de personalizar las propiedades listadas

Details			
	Number of Product	Description of Product	Quantity
			=
		and Teams	2
	2	Arco iris de lágrimas	1

There are 2 objects in list

**Figure 5.8 Efecto de @ListProperties en la interfaz de usuario de details**

La figura 2.8 también muestra como puedes usar la anotación @ListProperties para definir propiedades a mostrar en la interfaz de usuario de la colección inicialmente. Decimos “inicialmente” porque el usuario puede personalizar las propiedades a mostrar en la colección y así adaptar los datos de la colección a sus necesidades o preferencias. @ListProperties es fácil de usar, míralo en el listado 2.16.

#### Listado 2.16 @ListProperties define las propiedades a mostrar en la colección

```
@OneToMany(mappedBy="parent", cascade=CascadeType.ALL)
@ListProperties("product.number, product.description, quantity")
private Collection<Detail> details = new ArrayList<Detail>();
```

Como puedes ver, solo has de poner como valor para @ListProperties la

lista de la propiedades que quieres separadas por comas. Puedes usar propiedades calificadas, es decir, usar la notación del punto para acceder a las propiedades de referencias, tal como `product.number` y `product.description` en este caso.

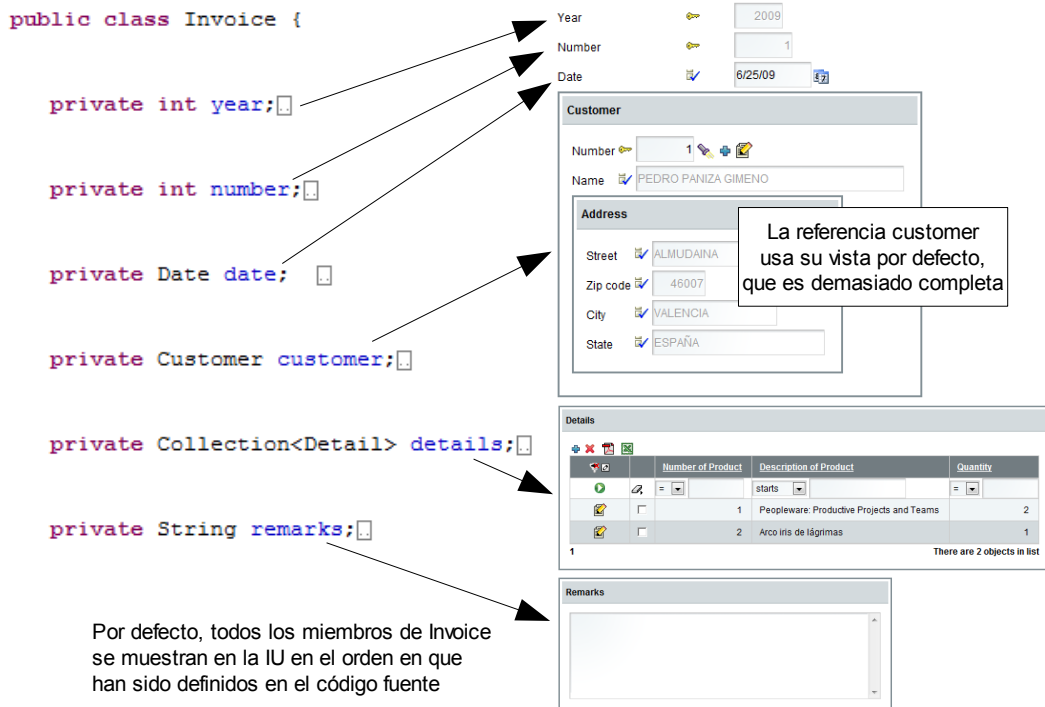
## **2.2 Refinar la interfaz de usuario**

¡Enhorabuena! Has finalizado tus clases del modelo del dominio, y tienes una aplicación funcionando. Tus usuarios pueden trabajar con productos, categorías, clientes e incluso crear facturas. En el caso de los productos, categorías y clientes la interfaz de usuario está bastante bien, aunque para factura todavía se puede mejorar un poco.

Ya has usado algunas anotaciones OpenXava para refinar la presentación, como `@DescriptionsList`, `@NoFrame` y `@ListProperties`. En esta sección usaremos más anotaciones de este tipo para dar a la interfaz de usuario de Invoice un mejor aspecto sin demasiado esfuerzo.

### **2.2.1 Interfaz de usuario por defecto**

La figura 2.9 muestra cómo es la interfaz de usuario por defecto para Invoice. Como ves, OpenXava muestra todos los miembros, uno debajo de otro, en el orden en que los has declarado en el código fuente. También ves como en el caso de la referencia `customer` se usa la vista por defecto de `Customer`.



**Figura 2.9** Interfaz de usuario por defecto generada automáticamente para Invoice

Vamos a hacer algunas pequeñas mejoras. Primero, definiremos la disposición de los miembros explícitamente, de esta forma podemos poner year, number y date en la misma línea. Segundo, vamos a usar una vista más simple para customer. El usuario no necesita ver todos los datos del cliente cuando está introduciendo la factura.

## 2.2.2 Usar @View para definir la disposición

Para definir la disposición de los miembros de Invoice en la interfaz de usuario has de usar la anotación @View. Es fácil, solo has de enumerar los miembros a mostrar. Mira el código en el listado 2.17.

### Listado 2.17 @View define la disposición de los miembros de Invoice

```

@View(members= // Esta vista no tiene nombre, por tanto será la vista usada por defecto
    "year, number, date;" + // Separados por coma significa en la misma línea
    "customer;" + // Punto y coma significa nueva línea
    "details;" +
    "remarks"
)
public class Invoice {

```

Mostramos todos los miembros de Invoice, pero

Year 2009 Number 1 Date 6/25/09

**Figura 2.10** Varios miembros en la misma línea



usamos comas para separar `year`, `number` y `date`, así son mostrados en la misma línea, produciendo una interfaz de usuario más compacta, justo como puedes ver en la figura 2.10.

### 2.2.3 Usar `@ReferenceView` para refinar la interfaz de referencias

Todavía necesitas refinar la forma en que la referencia `customer` se visualiza, porque visualiza todos los miembros de `Customer`, y para introducir los datos de una `Invoice` una vista más simple del cliente puede ser mejor. Para hacer esto, has de definir una vista `Simple` en `Customer`, y entonces indicar en `Invoice` que quieres usar esa vista `Simple` de `Customer` para visualizarlo.

Primero definamos la vista `Simple` en `Customer`, como muestra el listado 5.18.

#### Listado 2.18 Un vista `Simple` para `Customer`, con solo `number` y `name`

```
@View(name="Simple", // Esta vista solo se usará cuando se especifique "Simple"
      members="number, name" // Muestra únicamente number y name en la misma línea
)
public class Customer {
```

Cuando una vista tiene un nombre, como en este caso, esa vista solo se usa cuando ese nombre se especifica. Es decir, aunque `Customer` solo tiene esta anotación `@View`, cuando tratas de visualizar un `Customer` no usará esta vista `Simple`, sino la generada por defecto. Si defines una `@View` sin nombre, esa vista será la vista por defecto, aunque este no es el caso.

Ahora has de indicar que la referencia a `Customer` desde `Invoice` use esta vista `Simple`. Esto se hace mediante `@ReferenceView`. Mira el listado 5.19.

#### Listado 2.19 `@ReferenceView` para especificar la vista a usar por la referencia

```
@ManyToOne(fetch=FetchType.LAZY, optional=false)
@ReferenceView("Simple") // La vista llamada 'Simple' se usará para visualizar esta referencia
private Customer customer;
```

Realmente simple, solo has de indicar el nombre de la vista de la entidad referenciada que quieres usar.

Después de esto la referencia `customer` se mostrará de una forma más compacta, como se ve en la figura 2.11.

**Figura 2.11 Referencia a Customer usando su vista Simple**

Hemos refinado un poco la interfaz de Invoice.

## 2.2.4 Refinar la introducción de elementos de colección

Ya has hecho alguna mejora de la interfaz de usuario para la colección con `@ListProperties`. Aunque el aspecto de la colección cuando el usuario añade un nuevo elemento es todavía un tanto engorroso. Mira la figura 2.12.

La interfaz de usuario por defecto para introducir un nuevo Detail es aparatoso

Muestra todos los datos del Product, además dentro de un marco

Puede que la foto no sea necesaria aquí

Las observaciones se pueden omitir

Queremos una interfaz más simple para introducir una nueva línea. Con solo los datos imprescindibles, en la misma línea, sin marco anidados y con las etiquetas apropiadas

**Figura 2.12 Introducir una línea de Invoice es aparatoso, mejor algo más simple**

La manera de simplificar la entrada de la línea de la factura es usando `@View` y `@ReferenceView`, que por cierto ya conoces..

Empecemos definiendo una vista por defecto para la entidad `Detail`. Justo como en el listado 2.20.

### Listado 2.20 Definir una vista por defecto para Detail

```
@Entity
```

```
@View(members="product, quantity") // En la misma línea, ya que están separado por comas
public class Detail {
```

Como ves, cambiamos el orden, primero product, después quantity. Además, ponemos los dos en la misma línea. Todavía product se visualiza usando una vista demasiado grande. Puedes arreglar esto usando @ReferenceView, como muestra el listado 2.21.

#### Listado 2.21 Referencia a Product en Detail con @ReferenceView y @NoFrame

```
@ManyToOne(fetch=FetchType.LAZY, optional=true)
@ReferenceView("Simple") // Product se visualiza usando su vista Simple
@NoFrame // No se usa un marco alrededor de los datos de product
private Product product;
```

Nota que también usamos @NoFrame para eliminar el marco alrededor del producto, y así integrar visualmente la cantidad con el número de producto y descripción.

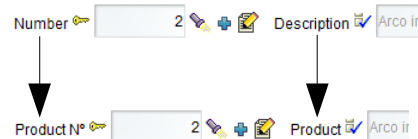
Por supuesto, necesitas una vista llamada Simple en la entidad Product. Veámosla en el listado 2.22.

#### Listado 2.22 Vista Simple en Product para ser usada desde Detail

```
@Entity
@View(name="Simple",
      members="number, description") // number y description en la misma línea
public class Product {
```

Ahora, el detalle contendrá en la misma línea el número de producto, la descripción del producto y la cantidad. Una interfaz limpia.

Ya que no usamos marco para product, las propiedades del producto se ven por el usuario como propiedades de Detail. Sería útil cambiar las etiquetas por defecto para number y description del producto para esta vista Simple para ser más claros de cara al usuario. Tal como muestra la figura 2.13.



**Figura 2.13 Etiquetas mas apropiadas para las propiedades del producto**

Establecer las etiquetas para las propiedades de una entidad en una vista concreta es fácil. Solo necesitas editar el archivo *Invoicing-labels\_en.properties* en */Invoicing/i18n*, y añadir las líneas del listado 2.23.

#### Listado 2.23 Entradas en Invoice-labels\_en.properties para las etiquetas

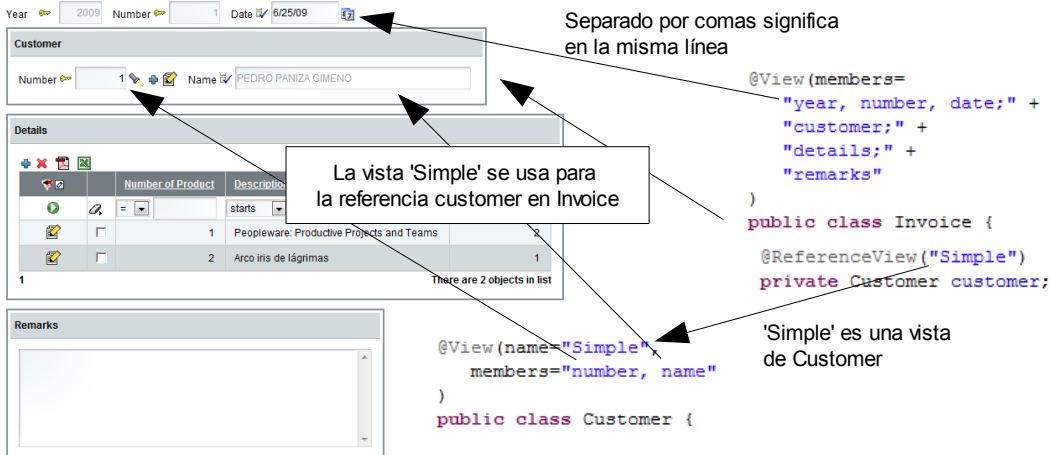
```
Product.views.Simple.number=Product Nº
Product.views.Simple.description=Product
```

Como ves, para definir las etiquetas se usa la notación del punto para la clave, definiendo primero el nombre de la entidad (Product en este caso), después

“views”, el nombre de la vista y el nombre de la propiedad. Obviamente, como valor se pone la etiqueta deseada.

### 2.2.5 La interfaz de usuario refinada

La figura 2.14 muestra el resultado de nuestros refinamientos en la interfaz de usuario de Invoice.



**Figura 2.14** Interfaz de usuario para Invoice con @View y @ReferenceView

Has visto lo fácil que es usar @View y @ReferenceView para obtener una interfaz de usuario más compacta para Invoice.

Ahora tienes una interfaz de usuario suficientemente buena para empezar a trabajar, y realmente hemos hecho poco trabajo para conseguirlo.

## 2.3 Desarrollo ágil

Hoy en día el desarrollo ágil ya no es una “una técnica nueva y rompedora”, sino una forma establecida de hacer desarrollo de software, es más, es la forma ideal de desarrollar software para muchos.

Si no estás familiarizado con el desarrollo ágil puedes echar un vistazo a [www.agilemanifesto.org](http://www.agilemanifesto.org). Básicamente, el desarrollo ágil favorece el uso de retroalimentación obtenida de un producto funcional sobre un diseño previo meticuloso. Esto da más protagonismo a los programadores y usuarios, y minimiza la importancia de los analistas y los arquitectos de software.

Este tipo de desarrollo necesita también un nuevo tipo de herramientas. Porque necesitas una aplicación funcional rápidamente. Tiene que ser tan rápido desarrollar la aplicación inicial como lo sería escribir la descripción funcional.

### 39 Lección 2: Modelar con Java

Además, necesitas responder a las peticiones y opiniones del usuario rápidamente. El usuario necesita ver sus propuestas funcionando en corto tiempo.

OpenXava es ideal para el desarrollo ágil no sólo porque permite un desarrollo inicial muy rápido, sino porque también te permite hacer cambios y ver su efecto instantáneamente. Veamos un pequeño ejemplo de esto.

Por ejemplo, una vez que el usuario ve tu aplicación y empieza a jugar con ella, se da cuenta que él trabaja con libros, música, programas y así por el estilo. Todos estos productos tienen autor, y sería útil almacenar el autor, así como ver los productos por autor.

Añadir esta nueva funcionalidad a tu aplicación es simple y rápido. Lo primero es crear una nueva clase para Author, con el código del listado 2.24.

#### Listado 2.24 Código para la entidad Author

```
package org.openxava.invoicing.model;

import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
import org.openxava.annotations.*;

@Entity
public class Author {

    @Id @GeneratedValue(generator="system-uuid") @Hidden
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid;

    @Column(length=50) @Required
    private String name;

    // Getters y setters
    ...
}
```

Ahora, añade el código del listado 2.25 a la ya existente entidad Product.

#### Listado 2.25 Referencia a Author desde Product

```
@ManyToOne(fetch=FetchType.LAZY)
@DescriptionsList
private Author author;

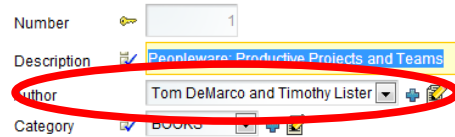
public void setAuthor(Author author) {
    this.author = author;
}

public Author getAuthor() {
    return author;
}
```

Así, tu entidad Product tiene una referencia a Author.

Realmente has escrito una cantidad pequeña de código. Para ver el efecto, solo necesitas construir tu proyecto (esto solo es pulsar Ctrl-B en tu Eclipse), lo cual es inmediato; actualizar el esquema de la base de datos, ejecutando la tarea ant `updateSchema`, solo dura unos pocos segundos. Ve al navegador y recarga la página con el módulo Product, y ahí verás, un combo para escoger el autor del producto, como muestra la figura 2.15.

¿Qué ocurre si el usuario quiere escoger un autor y ver todos sus productos? Está chupado. Solo has de hacer la relación entre Product y Author bidireccional. Ve a la clase Author y añade el código del listado 2.26.



**Figura 2.15 Referencia a Author desde Product**

#### Listado 2.26 Colección products en entidad Author

```
@OneToMany(mappedBy="author")
@ListProperties("number, description, price")
private Collection<Product> products;


public Collection<Product> getProducts() {
    return products;
}

public void setProducts(Collection<Product> products) {
    this.products = products;
}
```





Ahora pulsas Ctrl-B (para construir) y refresca tu navegador con el módulo Author. Escoge un autor y verás sus productos. Tienes algo parecido a la figura 2.16. Sí, añades una nueva colección, refrescas tu navegador y tienes una interfaz de usuario completa para manejarla.









En este caso no ha sido necesario actualizar el esquema, solo has de hacerlo cuando la estructura de la base de datos cambie, y esto normalmente ocurre cuando añades, quitas o renombas campos o relaciones en las entidades; o bien cuando creas nuevas entidades. El caso de la colección de productos es una excepción, porque la columna necesaria para esta relación ya estaba en la tabla de productos. De todas formas, no te preocupes mucho por esto, ante la duda, simplemente actualiza el esquema, no hace daño.

## 41 Lección 2: Modelar con Java

Name  Tom DeMarco and Timothy Lister

Products



		Number	Description	Price
		= 	starts 	= 
	<input type="checkbox"/>	1	Peopleware: Productive Projects and Teams	31.00
	<input type="checkbox"/>	3	Adrenaline Junkies and Template Zombies	35.00

1 There are 2 objects in list

**Figura 2.16** Interfaz de usuario para Author con una colección products

Esta sección ha mostrado el código completo y los pasos para hacer cambios y ver el resultado de una manera muy interactiva. Tus ojos han visto como OpenXava es una herramienta ágil, ideal para hacer desarrollo ágil.

## 2.4 Resumen

Esta lección te ha enseñado como usar simples clases de Java para crear una aplicación Web. Con solo escribir clases Java para definir tu dominio, obtienes una aplicación lista para usar. También, has aprendido como refinar la interfaz de usuario por defecto usando algunas anotaciones de OpenXava.

Efectivamente tienes una aplicación funcional con poco esfuerzo. Aunque esta aplicación “tal cual” puede servir como utilidad de mantenimiento o un prototipo, todavía necesitas añadir validaciones, lógica de negocio, comportamiento de la interfaz de usuario, seguridad y así por el estilo para convertir estas entidades que has escrito en una aplicación de gestión lista para tus usuarios.

Aprenderás estos temas avanzados en las siguientes lecciones.